Subobject-Oriented Programming

Marko van Dooren, Dave Clarke, and Bart Jacobs

iMinds-DistriNet, Department of Computer Science,
KU Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract. Classes are fundamental elements in object-oriented programming, but they cannot be assembled in a truly flexible manner from other classes. As a result, cross-cutting structural code for implementing associations, graph structures, and so forth must be implemented over and over again. Subobject-oriented programming overcomes this problem by augmenting object-oriented programming with subobjects. Subobjects can be used as buildings blocks to configure and compose classes without suffering from name conflicts. This paper gives an overview of subobject-oriented programming and introduces mechanisms for subobject initialization, navigation of the subobject structure in super calls, and subobject refinement. Subobject-oriented programming has been implemented as a language extension to Java with Eclipse support and as a library in Python.

1 Introduction

Class-based object-oriented programming enables programmers to write code in terms of abstractions defined using classes and objects. Classes are constructed from *building blocks*, which are typically other classes, using inheritance. This allows a new class to extend and override the functionality of an existing class. More advanced mechanisms such as multiple inheritance, mixins, and traits extend the basic inheritance mechanism to improve code reuse. Although becoming increasingly popular in mainstream languages, these mechanisms suffer from a number of problems: code repetition, conceptually inconsistent hierarchies, the need for glue code, ambiguity, and name conflicts.

Code repetition occurs because code cannot be sufficiently modularized in single and multiple inheritance hierarchies. For instance, many classes are based on high-level concepts such as associations (uni- or bi-directional relationships between objects) or graph structures, for example, in classes representing road networks. When implementing such classes, current best practice is to implement these high-level concepts with lots of cross-cutting, low-level boilerplate code. As a result, what is essentially the same code, is written over and over again. Attempts to improve reuse often result in conceptually inconsistent hierarchies, in which semantically unrelated code is placed in classes just to enable reuse. Often reuse is achieved by abandoning inheritance and instead delegating to auxiliary objects (delegates) that implement the desired functionality. This requires additional glue code to put the delegates in place and to initialise them. Even more glue code is required to handle overriding of delegate functionality in subclasses. Inheritance mechanisms that enable inheritance from multiple sources, such as multiple inheritance, mixins, and traits, cannot properly deal with methods that have the same name but come from different super classes or traits.

Modern programming languages offer some solutions to these problems, but these are inadequate for various reasons (see Section 2 for more detail). Composition mechanisms such as mixins [4] and traits [21, 19, 8, 7] cannot be used to turn the low-level code into reusable building blocks, because only one building block of a given kind can be used in a class. With non-virtual inheritance in C++ [22], the composer has no control over the inheritance hierarchy of the building block, and methods cannot always be overridden. With non-conformant inheritance in Eiffel [23], all methods of all building blocks must be separated individually. This requires a large amount of work and is error-prone. Aspect-oriented programming [15] does modularize cross-cutting code, but the cross-cutting code is of a different nature. In current aspect-oriented languages, the cross-cutting code of aspects augments the basic functionality of a system, whereas in the case of e.g. the graph functionality for road networks, the cross-cutting code defines the basic structure of the classes. While features such as intertype declarations in AspectJ [14] and wrappers in CaesarJ [1] can change the class structure, their purpose is to support aspects. As such, they do not provide the expressiveness to capture structural patterns that can be added multiple times to a single class, which is needed for a building block approach. And even if such expressiveness were supported, a class should not have to rely on aspects to define its basic structure. Together these problems indicate that the currently available building blocks for constructing classes are inadequate.

Subobject-oriented programming, the class composition mechanism described in this paper, addresses the limitations described above. Subobject-oriented programming augments object-oriented programming with subobjects that allow classes to be flexibly used as building blocks to create other classes. Subobjects do not introduce name conflicts, as different subobjects are isolated by default and can only interact after configuration in the composed class. Subobject members are truly integrated into the composed class and can be redefined in subclasses of the composed class. Subobject-oriented programming improves the *component relation* developed in previous work [26]; it has been used to make software transactional memory more transparent [24]. In this paper, we present an overview of subobject-oriented programming, along with the following new contributions:

- Language constructs for subobject initialization and navigation of the subobject structure in super calls.
- A simplified, object-oriented, approach to subobject configuration.
- A further binding mechanism to refine subobjects in a subclass.
- A formal models of the semantics of subobjects.
- Experimental validation: we have implemented subobject-oriented programming as a Java [12] extension called JLo [25], and as a Python 3 [20] library, and developed an extended example demonstrating how graph functionality can be implemented top of an existing association mechanism.

Organization: Section 2 presents an evaluation of existing reuse mechanisms. Section 3 defines subobject-oriented programming, including subobject customization and initialization. Section 4 shows additional examples of subobject-oriented programming. Section 5 discusses the implementation. Section 6 presents a semantic model of subobject-oriented programming, focusing on the semantics of method dispatch. Section 7 discusses related work, and Section 8 concludes.

2 Evaluation of Current Reuse Mechanisms

This section evaluates existing reuse techniques by trying to compose a simple class of radios from generic building blocks. The interface of class *Radio* is shown in Figure 1(a). The volume of a radio is an integer between 0 and 11, and the frequency is a float between 87.5 and 108.

```
class Radio {
        Radio(int, float)
  int   getVolume()
  void  setVolume(int)
  float getFrequency()
  void  setFrequency(float)
}
```

(a) The interface of *Radio*.

```
class BoundedValue
  <T extends Comparable<T>> {
       BoundedValue(T, T, T)
  T    getMax()
  T    getValue()
  void setValue(T)
  T    getMin()
}
```

(b) The interface of *BoundedValue*.

(c) A user interface for a radio.

```
class Slider {
  connect(BoundedValue value)
  ...
}
```

(d) The interface of *Slider*.

Fig. 1: The radio example.

The behavior of both the volume and frequency is similar in that both are numeric values that must remain within certain bounds. This behavior is encapsulated in *BoundedValue*, whose interface is shown in Figure 1(b). We want to implement *Radio* using two building blocks of this class. Note that *BoundedValue* itself could be implemented with three building blocks for the value and the upper and lower bounds, but we do not consider nested composition for this example as it does not add to the discussion.

To make a user interface for the radio, we want to use two *Slider* objects and connect them to the frequency and the volume building blocks *as if they were* separate objects.

The resulting class *Radio* should offer at least the same functionality as if it were implemented from scratch. External clients of *Radio* should see it via an interface similar to the one in Figure 1(a), and subclasses of *Radio* should be able to redefine its methods.

2.1 Aspect-Oriented Programming

In the original paper on aspect-orientation [15], aspects captured any kind of cross-cutting code. Current aspect-oriented languages, however, focus on aspects that modify the behavior of existing code. Although aspect-oriented languages provide features to

modify the structure of an existing class, such as intertype declarations in AspectJ [14] and wrappers in CaesarJ [1], they do not provide the expressiveness that is needed to add them multiple times to a single class. As such, the bounded values of *Radio* cannot be added with aspects.

But even if the required expressiveness would be added, the nature of the cross-cutting code makes aspect-oriented programming inappropriate for the job. In such an aspect-oriented language, the volume and frequency of the radio would be added to *Radio* in a separate aspect. The bounded values for the volume and the frequency, however, define the basic behavior of a radio, and should therefore be declared in class *Radio*.

2.2 Mixins

Mixin inheritance [4] provides a limited form of multiple inheritance. A mixin is a class with an abstract superclass. This super class can be instantiated differently in different contexts. A concrete class specifies its inheritance hierarchy by declaring a list of mixins, which is called a mixin composition. The mixin composition linearizes the inheritance hierarchy of a class by using the successor of each mixin in the list as the super class for that mixin. By eliminating diamond structures, the linearization mechanism automatically solves name conflicts. This, however, can cause a method to override another method with the same signature by accident.

Fig. 2 shows a Scala [18][1] example where we attempt to reuse class *BoundedValue* twice in class *Radio*. Class *BoundedValue* itself is omitted. Because of the linearization, it is impossible to use mixins to implement the volume and frequency of a radio in this fashion. All methods inherited through the clause ***with*** *BoundedValue[Float]* would be overridden by or cause conflicts with the (same) methods inherited through the clause ***with*** *BoundedValue[Integer]*.

```
class Radio extends BoundedValue[Int] with BoundedValue[Float]
```

Fig. 2: An attempt to implement *Radio* with mixins.

2.3 Traits

A trait [21] is a reusable unit of behavior that consist of a set of methods. A class uses a trait as a building block through *trait composition*, and resolves name conflicts explicitly using trait operators such as *alias* or *exclude*. The *flattening property* of traits states that the behavior of a trait composition is the same as if the trait code was written in the composing class. Lexically nested traits [7] improve upon the original traits by allowing a trait to have state and providing a lexical mechanism for controlling the visibility of the methods provided by a trait. Lexically nested traits are implemented

[1] Note that Scala does not support traits; it uses the keyword *trait* for mixins.

in the AmbientTalk [7] programming language. For the sake of simplicity, the example code uses a hypothetical language TraitJava, which adds lexically nested traits to Java.

Fig. 3 shows an attempt to implement a radio using trait composition in TraitJava. Trait composition is done by *using* the trait. Class *Radio* uses two *BoundedValue* traits to model the volume and the frequency. Aliasing is used to resolve the name conflicts caused by using two traits with methods that have the same name.

```
class Radio {
  uses BoundedValue<Int> {
    alias value -> volume, max -> maxVolume,
          min -> minVolume, setValue -> setVolume
  }
  uses BoundedValue<Float> {
    alias value -> frequency, max -> maxFrequency,
          min -> minFrequency, setValue -> setFrequency
  }
}
trait BoundedValue<T> {
  T _value;
  T value() {return _value;}
  void setValue(T t) {
    if(t >= min() && t <= max()) {_value = t;}
  }
  T max() {...}
  T min() {...}
  ...
}
```

Fig. 3: Traits in the hypothetical language TraitJava.

While at first sight this code seems to do what we need, it does not actually work. The fields of both bounded values are properly isolated from each other due to lexical scoping, but the methods are not. Remember that the flattening property ensures that the code behaves as if it were written directly inside the using class. Therefore, the *setVolume* and *setFrequency* methods have an identical implementation and thus use the same methods to obtain the upper and lower bounds. As a result, both *setVolume* and *setFrequency* try to invoke methods named "min" and "max" on a radio at run-time. But a radio does not even have methods with names "min" or "max" because a trait alias is not a true alias. Instead, an alias only inserts a delegation method that invokes the "aliased" method on the trait object. Note that even if the methods of one of one of the two traits were not aliased, the other bounded value object would always use the bound methods of the former.

Trait-based metaprogramming [19] is similar to non-conformant inheritance in Eiffel, which is discussed in the next section. Trait-based metaprogramming is discussed in more detail in the related work section.

2.4 Non-conformant Inheritance in Eiffel

Eiffel [23] and SmartEiffel 2.2 [6] support non-conformant inheritance – inheritance without subtyping – to insert code into a class through the *insert* relation. As with the regular subclassing relation, multiple inheritance and repeated inheritance (inheriting from the same class more than once) are allowed.

Fig. 4 shows how class *RADIO* implements its volume and frequency functionality using repeated non-conformant inheritance. Constructors are omitted to save space. The methods and fields of *BOUNDED_VALUE* are duplicated by giving them distinct names using renaming. In case of repeated inheritance, self invocations in Eiffel are bound *within the inheritance relation of the current method*. Suppose that the constructor of *BOUNDED_VALUE* (*make*) invokes *set_value*. When *make_vol* is executed in the constructor of *RADIO*, the *set_value* call is bound to *set_vol*, which sets the *vol* field. This is exactly the behavior that we need.

```
class RADIO
  inherit {NONE}
    -- All members must be separated explicitly.
    BOUNDED_VALUE
      rename make as make_freq,
             set_value as set_freq, value as freq,
             max as max_freq, min as min_freq end
    BOUNDED_VALUE
      rename make as make_vol,
             set_value as set_vol, value as vol,
             max as max_vol, min as min_vol end
end
class BOUNDED_VALUE[T <: COMPARABLE[T]]
  min, value, max: T
  set_value(val: T) do
    if min <= val and val <= max then value := val
end end end
```

Fig. 4: Implementing *Radio* in Eiffel.

By default, members that are inherited via multiple inheritance paths form a single definition. Therefore, if we want to use the insert relation to create *Radio*, all members of the volume and frequency must be renamed individually in order to duplicate them. This not only requires a lot of work, but is also error-prone because no compile error is reported when some members are forgotten. For example, if *max* is not duplicated, the volume and frequency share the same upper bound. Another problem is that it is not possible to use two *Slider* objects for the user interface because neither the volume, nor the frequency can be used as *BOUNDED_VALUE* objects.

2.5 Manual Delegation

The *Radio* class can also be built using two *BoundedValue* objects and manually writing the delegation code, as shown in Fig. 5. Both bounded values are properly separated, but writing the delegation code is cumbersome and error-prone. Furthermore, it requires anticipation: at least one special constructor is needed in each class to allow subclasses to change the behavior of the bounded values. In the example, class *EventRadio* uses an *EventBoundedValue* to send events when the volume is changed. If the special constructor is forgotten, neither the volume nor the frequency can be customized in subclasses.

By exposing the *BoundedValue* objects via methods *volume()* and *frequency()*, the user interface for the radio can be made by connecting two *Slider* objects to the *BoundedValue* objects for the volume and the frequency.

```
class Radio {
  BoundedValue<Int> _v;
  BoundedValue<Int> volume() { return _v;}
  final Int getVolume() {return _v.getVal();}
  final void setVolume(Int v) {_v.setVal(v);}
  BoundedValue<Float> _f;
  BoundedValue<Float> frequency() { return _f;}
  final Float getFreq() {return _f.getVal();}
  final void setFreq(Float f) {_f.setVal(f);}

  Radio(Int v, Float f) {this(null,volume,null,f)}
  // The BoundedValue objects must be changable.
  Radio(BoundedValue<Int> subV, Int v,
        BoundedValue<Float> subF, Float f){
    if (subV != null) _v = subV;
    else _v=new BoundedValue<Int>(0,v,11);
    if (subF != null) _f = subFrq;
    else _f=new BoundedValue<Float>(87.5,f,108);
  }
}
class EventRadio {
  EventRadio(Int v)
    {super(new EventBoundedValue<Int>(0,v,11);}
  EventRadio(EventBoundedValue<Int> subV, Int v
        EventBoundedValue<Float> subF, Float f)
    {super(subV,v,subF,f);}
}
```

Fig. 5: Manual delegation in Java.

2.6 Scala Objects

On the surface it appears that object declarations in Scala can be used. Consider for example the code in Fig. 6. The *BoundedValue* value objects for the volume and frequency are completely separated, and delegation methods are defined. A subclass *EventRadio*, however, cannot change the objects to *EventBoundedValue* objects since objects cannot be overridden in Scala. While the delegation methods can be overridden, that does not affect the behavior of the *volume* or *frequency* objects.

```
class Radio {

  // objects for volume and frequency
  object volume extends BoundedValue
  object frequency extends BoundedValue

  // providing aliases for subobject methods
  def getVolume = volume.getValue
  def setVolume = volume.setValue
  def getFrequency = frequency.getValue
  def setFrequency = frequency.setValue
}

class EventRadio extends Radio {
  // does not override volume.getValue
  override def getVolume = ...

  // generates compile error
  object volume extends EventBoundedValue
}
```

Fig. 6: Implementing *Radio* with objects in Scala.

2.7 Summary

Even though the radio example is simple, none of the typical reuse techniques is able to maintain that simplicity in the implementation. Mixins and traits cannot be used at all because they do not offer support for using multiple building blocks of the same type. Non-conformant inheritance allows easy customization without anticipation, but requires a lot of work to separate the building blocks, and does not allow them to be used as if they were separate objects. With manual delegation, the situation is exactly the other way around. Isolating the building blocks and using them as separate objects is easy, but customization is less elegant and requires anticipation.

3 Subobject-Oriented Programming

Subobject-oriented programming augments object-oriented programming with subobjects, which allow developers to capture cross-cutting structural boilerplate code in a class and then use it as a configurable building block to create other classes. A subobject can be seen as a combination of inheritance and delegation. It combines the ability to have isolated and accessible building blocks, as provided by delegation, with the ability to easily modify their behavior without anticipation and integrate them in the interface of the composed class, as with inheritance.

Subobject members can be exported to the surrounding class, and customized either in the subobject itself or in the surrounding context. Subobjects can be refined in subclasses of the composed class, they can be treated as real objects.

Fig. 7 shows the syntax of subobjects. Subobjects are *named* members of the composed class. Type T is the *declared superclass* of the subobject. Members defined in the body of a subobject may also redefine members of T. Within the body of a subobject, the *this* expression refers to the subobject itself. The *outer* expression refers to the (sub)object that directly encloses the subobject. The value of *outer* is equal to the value of *this* in the enclosing (sub)object. Similar to invocations on *this*, invocations on *outer* are bound dynamically. Section 3.2 explains *export* clauses. Section 3.3 explains subobject refinement, *overrides* and *refines* clauses, and *super* calls. Section 3.4 explains subobject initialization. Note that we use the Scala syntax for methods throughout the paper since it is more concise than the syntax of Java.

Class	::= **`class`** *T* **`extends`** *T* $\overline{\textbf{\texttt{implements}}\ T}\{\overline{Member}\}$
Member	::= *Method* | *Field* | *Subobject* | *Export*
Subobject	::= **`subobject`** *Id T [Body]*
Export	::= **`export`** $\overline{Path}$ *[***`as`** *Id]*
Refines	::= *Id* **`refines`** *Path*
Overrides	::= *Id* **`overrides`** *Path*
Path	::= $\overline{Id}$
SubobjectInit	::= **`subobject`**.*Path*($\overline{e}$)
SuperCall	::= *[Prefix .]* **`super`** *[. Path]* .*m*($\overline{e}$)
Prefix	::= $\overline{\textbf{\texttt{outer}}}$ | *Path* | $\overline{\textbf{\texttt{outer}}}$.*Path*

Fig. 7: Syntax for subobjects.

3.1 Subobject Basics

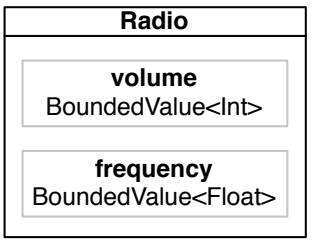
Fig. 8 shows how *Radio* implements its volume and frequency with subobjects named *volume* and *frequency*. These subobjects have declared superclasses *BoundedValue*<*Integer*> and *BoundedValue*<*Float*> respectively. The interface of *Radio* does not yet contain the setters and getters for the volume and the frequency.

```
class Radio {
  subobject volume BoundedValue<Integer>;
  subobject frequency BoundedValue<Float>;
}

class BoundedValue<T> {
  BoundedValue(T min, T val, T max) {...}

  T getValue() = value;
  void setValue(T value) {
    if(value >= getMin() && value <= getMax())
      {this.value = value;}
  }
  T value;

  // similar code for the min and max values
}
```

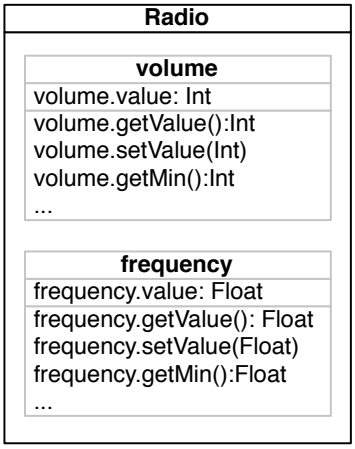
Fig. 8: Using subobjects for the volume and the frequency.

The diagram in Fig. 9a shows the class diagram for *Radio*. A subobject is visualized as a box inside the composed class because it is used as a building block for that class. The declared super class of the subobject is shown underneath its name.

Subobjects are isolated by default, without requiring renaming or exporting. The diagram in Fig. 9b shows a flattened view of the members of the *Radio* class of Fig. 8. The members that are directly available in the interface of *Radio* are shown in bold. Subob-



(a) The class diagram of *Radio*.



(b) A flattened view of the members of *Radio*.

Fig. 9: Visualization of class *Radio*.

ject *volume* introduces the following members into *Radio*: *volume.getValue*, *volume.setValue*, *volume.value*, *volume.getMin*, and so forth. Similarly, subobject *frequency* introduces *frequency.getValue*, *frequency.setValue*, and so forth. This avoids an explosion of name conflicts in the enclosing class.

Within the context of a subobject, *this* binds to that subobject. Therefore, method calls and field accesses executed in the context of a subobject are bound within that subobject. For example, to verify that the value of a *BoundedValue* is not set to an invalid value, the *setValue* method of class *BoundedValue* must call *getMin* and *getMax* to obtain the bounds. During the execution of *volume.setValue*, these calls are bound to *volume.getMin* and *volume.getMax* respectively.

Using Subobjects As Real Objects The name of a subobject allows it to be used as a real object whose type is a subtype of its declared superclass. The subtype reflects any customization done in the subobject body, such as using a more specific return type. Fig. 10 illustrates how a subobject is used as a real object. Invoking *r.volume* returns an object of type *Radio.volume*, which is a subtype of *BoundedValue<Integer>*. This allows the subobject to be connected to a slider of the user interface.

```
class Slider<T extends Number> {
  BoundedValue<T> _model;
  void connect(BoundedValue<T> bv) {...}
}
Radio r = new Radio();
Slider<Int> vs = new Slider<Int>();
vs.connect(r.volume); // similar for frequency
```

Fig. 10: Using subobjects as real objects.

Nested Subobjects Nested subobjects are also available in the composed class. The code in Fig. 11 shows a part of the radio example with nested subobjects. Note that this version is incomplete as it does not perform any bounds checks. The example merely serves to illustrates nesting. In Sect. 3.2 we show how subobject members can be made available directly in the interface of the composed class. In Sect. 3.3 we show how subobject members can be overridden to enforce the upper and lower bounds of *BoundedValue*. Class *BoundedValue* now uses three *Property* subobjects with names *value*, *min*, and *max* for the value and the bounds. Class *Property* has methods *getValue* and *setValue* and field *value*. In this case, *BoundedValue* does not offer *getValue* and *setValue* directly in its interface.

The diagram in Fig. 12 shows the members of *Radio* in this scenario. Class *Radio* now has members *volume.value.getValue*, *volume.min.getValue*, *volume.max.getValue*, and so forth. Note that a developer is not confronted with a large number of members

```
class Radio {
  subobject volume BoundedValue<Integer>;
  subobject frequency BoundedValue<Float>;
}

class BoundedValue<T> {
  subobject min Property<T>;
  subobject value Property<T>;
  subobject max Property<T>;
}

class Property<T> {
  T value;
  T getValue() = value;
  void setValue(T t) {value = t;}
}
```

Fig. 11: Part of the radio example with nested subobjects.

when looking at the interface of *Radio*. Only the members shown in bold are directly accessible. The other can only be accessed via the subobjects.

Radio
volume
volume.min
volume.min.value
volume.min.getValue()
volume.min.setValue(int)
volume.value
volume.value.value
volume.value.getValue()
volume.value.setValue(Int)
volume.max.value
volume.max.getValue()
volume.max.setValue(int)
…
frequency
...

Fig. 12: A flattened view of the members of *Radio* (nested version).

3.2 Exporting Subobject Members

While a subobject can be accessed as an object, it is more convenient if commonly used members are available directly in the composed class. In addition, it is desirable to give such members names that are appropriate for the composed class. Subobject members are added to the interface of the composed class using *export* clauses. In Fig. 13, the

getter and setter methods for the volume and the frequency *BoundedValue*s are exported to *Radio* as *getVolume*, *setVolume*, *getFrequency*, and *setFrequency*. Whether the getter and setter methods are implemented directly in *BoundedValue* or exported to *BoundedValue* from nested subobjects makes no difference for class *Radio*.

```
class Radio {
  subobject volume BoundedValue<Int> {
    export getValue as getVolume,
           setValue as setVolume;
  }
  subobject frequency BoundedValue<Float> {
    export getValue as getFrequency,
           setValue as setFrequency;
  }
}
```

Fig. 13: Adding getters and setters to *Radio*.

The class diagram of *Radio* with the exported members is shown in Fig. 14. The getter and setter methods of the *volume* subobject can not be invoked directly on an object of type *Radio* via *getVolume* and *setVolume* respectively. The notation $m < f$ indicates that member f from the subobject is exported to the composed class as m. The new name is written at the left side to improve the readability of the interface of the composed class.

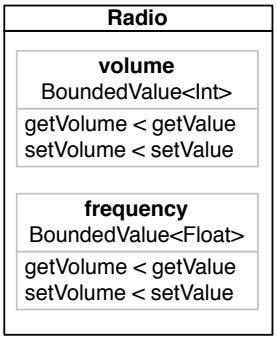


Fig. 14: The class diagram of *Radio* with exported members.

An export clause ***export*** `path.d` in subobject *s* makes member *s.path.d* accessible via name *d* in the enclosing scope, so long as doing so does not create name conflicts. The form ***export*** `path.d` ***as*** `Id` can also be used to give a new name to the exported path. In both cases, the member is still available via its original path (`s.path.d`).

The alias defined by an export clause cannot be broken, as shown in Fig. 15. Class *BrokenRadio* overrides *setVolume* to set the value to the opposite value of the scale. Because of the aliasing, *volume.setValue* is also overridden when *setVolume* is overridden. Thus regardless of whether the client changes the volume via *setVolume* or via *volume.setValue*, the effect is always the same.

```
class BrokenRadio extends Radio {
  void setVolume(Int vol) {super.setValue(11-vol);}
}
BrokenRadio br = new BrokenRadio();
br.setVolume(1); // write directly
//read through subobject equals direct read
assert(br.volume.getValue() == br.getVolume());
br.volume.setValue(2); //write via subobject.
//read through subobject equals direct read
assert(br.volume.getValue() == br.getVolume());
```

Fig. 15: Overriding cannot break aliases.

Export clauses provide the best of two worlds: ease of use and reuse. The composed class can provide an intuitive and uncluttered interface. Classes meant to be used as subobjects can provide a lot of functionality without cluttering the composed classes. Members that are not exported can still be accessed by using the subobject as a separate object whose type is its declared superclass, as illustrated in Fig. 16. In other approaches, such members are no longer available, resulting in code duplication.

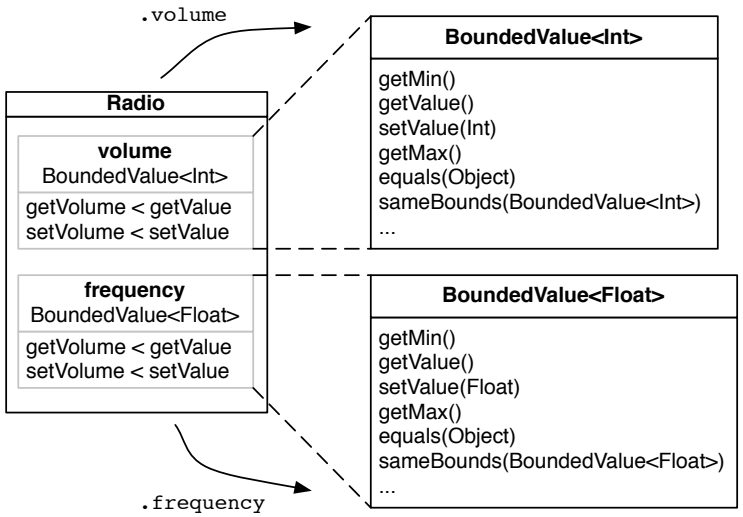


Fig. 16: Zooming in on a subobject.

3.3 Customizing Subobjects

Methods of a subobject can be overridden in its body. The return type is covariant and the parameter types are invariant. Otherwise the code in the subobject may break.

```
class BoundedValue<T extends Comparable<T>> {
 subobject value Property<T> {
   export getValue, setValue;
   boolean isValid(T t) =
     (t != null &&
     outer.getMin() <= t && t <= outer.getMax())
 }

 subobject min Property<T> {
   export getValue as getMin, setValue as setMin;
   boolean isValid(T t) =
     (t!= null && t <= outer.getValue())
 }

 subobject max Property<T> {
   export getValue as getMax, setValue as setMax;
   boolean isValid(T t) =
     (t!= null && outer.getValue() <= t)
 }
}
```

Fig. 17: Overriding subobject members.

Suppose that *BoundedValue* is implemented using three *Property* subobjects for the value and the bounds, as shown in Fig. 17. The setter of *Property* uses *isValid* to validate the given value. Class *BoundedValue* redefines the *isValid* methods of the three subobjects to ensure that the value will be between the upper and lower bounds. The subobjects are isolated by default, thus *outer* and *export* are used to cross the boundaries of the subobjects.

Fig. 18 shows the class diagram of *BoundedValue*. The members that are overridden in the subobject are shown in a separate area.

The diagram in Fig. 19 shows the lookup table of *Property* and a part of the lookup table of *BoundedValue*. For each subobject, there is an additional lookup table. Field reads and writes are bound dynamically. Note that the lookup table for a subobject contains a new entry for each field of the declared superclass. A new entry is needed because the position of the fields of a subobject in the memory layout of the object of the outer class is specific for each outer class. The *getValue* and *setValue* methods of *BoundedValue.value* point to the corresponding implementations in *BoundedValue*, while its *isValid* method uses the overriding implementation *M_x*.

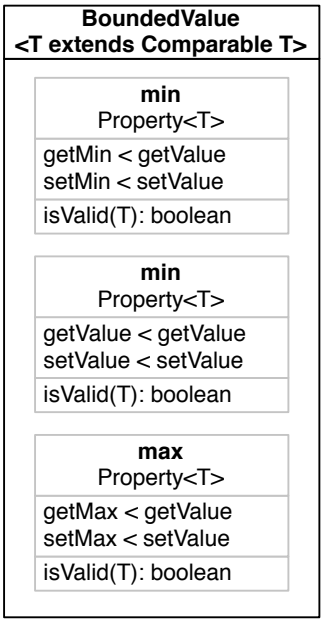


Fig. 18: The class diagram of *BoundedValue*.

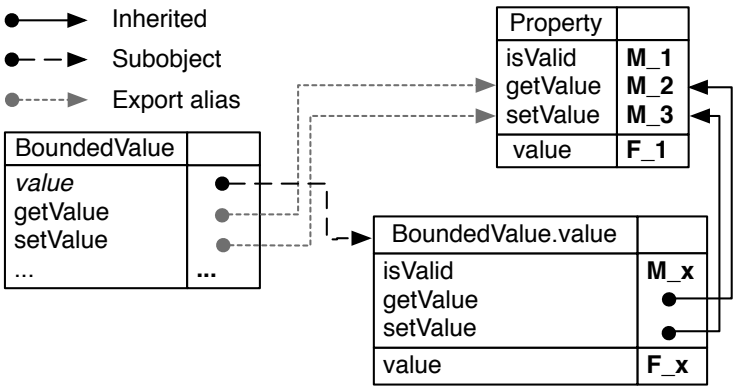


Fig. 19: Part of the lookup tables of *BoundedValue*.

Overriding in an Enclosing Scope In some cases, the overriding method cannot be written directly in the subobject, for example, when a programmer wants to override a method of a subobject using a method the outer class inherits from a superclass. Another example is when methods of different subobjects need to be joined to share the same implementation. In these situations, an *overrides* clause written in an outer scope can be used to achieve the desired effect.

```
class StereoRadio {
  subobject frequency BoundedValue<Float> {
    export getValue as getFrequency,
           setValue as setFrequency;
  }
  subobject left BoundedValue<Int> {
    export getValue as getLeftVol,
           setValue as setLeftVol;
  }
  subobject right BoundedValue<Int> {
    export getValue as getRightVol,
           setValue as setRightVol;
  }
  setMaxVolume overrides left.setMax;
  setMaxVolume overrides right.setMax;
  void setMaxVolume(Int v) {
    left.super.setMax(v);
    right.super.setMax(v);
  }
  isValidMaxVolume overrides left.max.isValid;
  isValidMaxVolume overrides right.max.isValid;
  void isValidMaxVolume(Int v) =
    left.max.super.isValid(v) && right.max.super.isValid(v)
}

StereoRadio r = new StereoRadio();
// Equivalent ways of setting the maximum volume
r.setMaxVolume(1);
r.left.setMax(1);
r.right.setMax(1);

// Both maximum volumes are always the same.
invariant(r.left.getMax() == r.right.getMax());
```

Fig. 20: Joining parts of two subobjects.

Suppose that we need a stereo radio for which the minimum and maximum volume of the left and the right channel are always the same. The code in Fig. 20 shows how this is implemented by joining the *BoundedValue* subobjects for both maximum volumes to-

gether; the code for the minimum volume is similar. *StereoRadio* defines a new setter for the maximum volume which invokes the overridden methods of both subobjects to set the maximum volumes. The overrides clauses specify that *setMaxVolume* overrides the *setMax* methods of both subobjects. Similar to export clauses, an overrides clause defines an alias relation between method names that cannot be broken. Therefore, the maximum volume of both channels is changed regardless of whether it is changed via *setMaxVolume* or via one of the subobjects. Without the ability to override methods in the outer scope, these methods would have to be overridden in the subobjects, resulting in code duplication. The diagram in Fig. 21 shows the class diagram of *StereoRadio*. The $>$ symbol represents an override clause that declares that the left-hand side overrides the right-hand side.

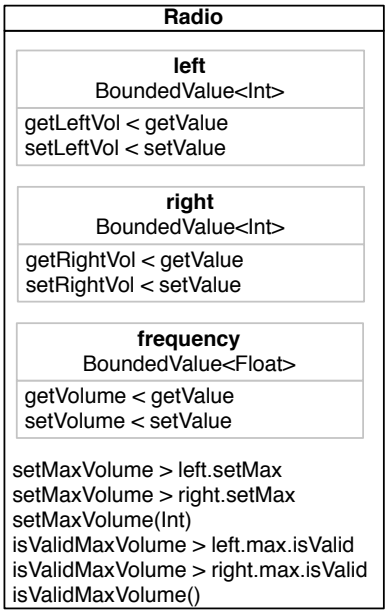


Fig. 21: The class diagram of *StereoRadio*.

Merging Fields The overrides clause can also be used to join fields of nested subobjects. For example the bounds of left and right volume of the stereo radio can also be joined by overriding *isValid* as in Fig. 20 or by merging the fields of the bounds instead. This alternative is illustrated in Fig. 22.

```
class StereoRadio {
  maxVolume overrides left.max.value;
  maxVolume overrides right.max.value;
  Integer maxVolume;
}
```

Fig. 22: Merging fields of two subobjects.

Subobject Refinement As subobjects are class members, they can also be modified in a subclass. Contrary to methods, subobjects are not overridden but are instead *refined*. This is similar to refinement (or further binding) of nested classes in Beta [17] and gbeta [10]. Our previous approach [26] allowed subobjects to be completely overridden, but this was fragile and required code duplication.

```
class EventRadio extends Radio {
  subobject frequency EventBoundedValue<Float>;
}
```

Fig. 23: Changing the declared superclass of a subobject.

Suppose that we want to create a subclass of *Radio* that sends events when the bounds or the value of the frequency are changed. Fig. 23 shows how this can be implemented using subobject refinement. We assume that *EventBoundedValue* is a subclass of *BoundedValue* that sends events. Class *EventRadio* refines the *frequency* subobject of *Radio* by changing its declared superclass to *EventBoundedValue*. The export clauses are inherited from *Radio.frequency*.

When subobject *t* refines subobject *s*, all members of *s* are inherited by *t*. A member *x* defined in the body of *t* overrides or refines a member of *s*, depending on whether *x* is a subobject or not. For export clauses, the existing name mapping cannot be changed; only additional mappings can be added. The declared superclass of *t* is equal to the declared superclass of *s*, unless *t* specifies its own declared superclass (*T*), in which case *T* must be a subtype of *S*.

Suppose now that we want to send an event only when the actual volume changes. In this case, we refine only the nested *value* subobject of the *volume* subobject. This is shown in Fig. 24. The *frequency* subobject inherits its declared superclass (*BoundedValue*<*Float*>) from the *frequency* subobject of *Radio*. Assume that *EventProperty* is a subclass of *Property* that sends events. The diagram in Fig. 25 shows how nested refinement is visualized in a class diagram. The subtype relations between the types of the subobjects involved in the refinement are not shown because that would clutter the diagram.

```
class EventRadio extends Radio {
  subobject frequency {
    subobject value EventProperty<Float>;
  }
}
```

Fig. 24: Nested subobject refinement.

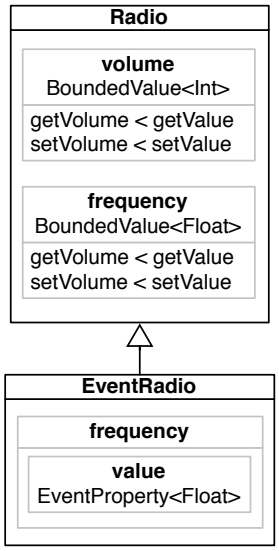


Fig. 25: A class diagram of nested refinement.

Super Calls Subobject refinement gives rise to a form of multiple inheritance because a subobject inherits members from the refined subobject and possibly a new declared superclass. Conflicts are detected using the *rule of dominance*, as used in C++ and Eiffel. If a subobject would inherit two different definitions for a member, then it must provide a new definition to resolve the conflict.

Suppose for example that *Radio.frequency* overrides *setValue* from its declared superclass *BoundedValue* to check whether the current frequency matches a pre-programmed channel. Class *EventBoundedValue* also overrides *setValue* to send events, which means that *EventRadio.frequency* has two candidate *setValue* methods. This conflict is resolved by defining a new *setValue* method as shown in Fig. 26.

Overriding methods typically use super calls to access overridden implementations. Because of the multiple inheritance, super calls must be disambiguated. Super calls therefore have the form `prefix.super.suffix.m(args)`. The semantics of a super call is as follows. The suffix and the method call are looked up in the scope determined by the prefix. If the prefix ends with a path (*Path* or `outer`.*Path*), then the scope is that subobject. If the prefix is empty or `outer`, the scope is the superclass of the enclosing scope referenced by the prefix. For a subobject, the superclass is the declared superclass. All binding in a super call is static.

In Fig. 26, expression `super.setValue(val)` invokes the *setValue* method of declared superclass *EventBoundedValue*, and `outer.super.frequency.setValue(val)` invokes the *setValue* method of *Radio.frequency*. In the super call, `outer` jumps to the enclosing class (*EventRadio*), `super` jumps up to *Radio*, and finally *frequency* jumps inward to the *frequency* subobject. In this case, the value is set twice, but we chose this implementation to illustrate how to access a refined subobject with a super call.

```
class Radio {
  subobject frequency BoundedValue<Float> {
    void setValue(Float val) {
      super.setValue(val);
      checkForPreProgrammedStation();}
  }
}
class EventRadio extends Radio {
  // refinement changes declared superclass
  subobject frequency EventBoundedValue<Float> {
    void setValue(Float val) {
      super.setValue(val);
      outer.super.frequency.setValue(val);
  }
}
```

Fig. 26: Accessing overriding methods.

Note that if *setValue* was overridden in *Radio* instead of in *Radio.frequency*, there would still be a conflict. Even though the *setValue* method would then not lexically be in the *frequency* subobject, it would still be the *setValue* method of *frequency* and thus cause a conflict with *EventBoundedValue.setValue*.

Refining Subobjects in an Enclosing Scope Similar to methods and fields, a subobject can also be redefined in a scope that encloses the subobject. A redefining subobject in an outer scope inherits all regular members from all redefined subobjects, but export clauses are not inherited. All redefined subobjects are joined into a single subobject.

We illustrate subobject refinement in an outer scope for the stereo radio example from Section 3.3. Remember that the bounds of both volume channels should always be the same. The previous approach (Fig. 20) joined the *setMax* methods from the *left* and *right* subobjects. This is unwieldy if many methods need to be joined. An alternative approach is to join the nested *Property* subobjects that represent the bounds. Fig. 27 shows how the nested subobjects for the bounds of the volumes can be joined. The *refines* clauses join the nested *max* subobjects of both channels.

The example in Fig. 27 illustrates the need to customize the rule of dominance for subobjects. Remember from Fig. 17 that *BoundedValue* overrides the *isValid* methods of its three *Property* subobjects to do the bounds check. As such, the *isValid* method of the *max* subobject of *BoundedValue* is more specific than *Property.isValid*. Using the traditional rule of dominance, no conflict would be reported because the *isValid* methods from both *left.max* and *right.max* originate from the same definition in *BoundedValue*. The behavior of these methods, however, is not at all the same. Invoking *left.max.isValid(val)* checks whether *val* is not smaller than the left volume, whereas *right.max.isValid(val)* checks whether *val* is not smaller than the right volume. This means that there are actually two most specific candidates instead of one. Therefore, both methods are overridden by a unique most specific version.

```
class Radio {
  subobject left BoundedValue<Int> {
    export getValue as getLeftVol,
           setValue as setLeftVol;
  }
  subobject right BoundedValue<Int> {
    export getValue as getRightVol,
           setValue as setRightVol;
  }

  // join both upper bounds
  maxVolume refines left.max;
  maxVolume refines right.max;
  subobject maxVolume Property<Int> {
    // This method must be redefined because
    // the original definition captures its
    // context.
    boolean isValid(Int val) =
      (outer.left.max.super.isValid(val) &&
       outer.right.max.super.isValid(val))
  }
  // similar for the minimum volume
}

Radio r = new Radio();
// equivalent ways of setting the max volume
r.maxVolume.setValue(1);
r.left.setMax(1);
r.right.setMax(1);

// both channels always have the same max value
invariant(r.left.getMax() == r.right.getMax());
```

Fig. 27: Refining nested subobjects in the composed class.

More formally, when member *m* is redefined in a subobject or in an enclosing scope, the redefinition of *m* can access all elements of the composed class *T* that contains the subobject. Therefore, the new definition of *m* depends on *T*. Suppose that subobject *s* refines multiple nested subobjects $\overline{x}.t$ of type *T*. Each member m_i of nested subobject $x_i.t$ depends on x_i and thus has a behavior that is potentially different from all m_j with $i \neq j$. As a result, no m_i can be selected automatically as *the* version of *m*. Therefore all member m_i conflict with each other in the context of *s*.

3.4 Initialization of Subobjects

During the construction of an object, its subobjects must be initialized as well. Initialization of a subobject is similar to a traditional super constructor call. No additional object is created, but the initialization code is executed on the new object of the inheriting class. In case of a subobject initialization call, however, the initialization code is executed on the *part* of the new object of the composed class that corresponds to the subobject. Syntactically, a subobject constructor call consists of the keyword ***subobject*** followed by the name of the subobject and the arguments passed to the constructor. Subobject constructors must be invoked directly after the super constructor calls. If the class of a subobject has a default constructor, no explicit subobject constructor call is required for that subobject. A subobject inherits all constructors from its declared superclass, but it cannot define constructors itself.

Consider the example in Fig. 28. To initialize its subobjects, the constructor of *Radio* performs two *subobject constructor calls*. Both calls invoke the same constructor of *BoundedValue*, but each call operates on a different part of the *Radio* object. Similarly, the constructor of *BoundedValue* invokes the constructor of *Property* for each of its three subobjects.

```
class Radio {
  Radio(Integer vol, Float freq) {
    // initialize the volume subobject
    subobject.volume(0,vol,11);
    // initialize the frequency subobject
    subobject.frequency(87.5,freq,108);
  }

  subobject volume BoundedValue<Integer> {...}
  subobject frequency BoundedValue<Float> {...}
}
```

Fig. 28: Initializing the subobjects of a radio.

Initialization of Refined Subobjects If a subobject is refined, initialization is more complicated. Whether the original subobject constructor calls remain valid depends on whether the declared superclass of the subobject has changed.

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {super(0,freq);}
  subobject volume {Float getValue() = 11}
}
```

Fig. 29: The refined subobject does not change the declared superclass.

We explore the different scenarios using a special class of radios for teenagers, whose volume is always set to the maximum. The first way to implement *TeenagerRadio* is to override the getter for the volume by refining the *volume* subobject and overriding *getValue*, as shown in Fig. 29. In this case, *TeenagerRadio.volume* is an anonymous subclass of *Radio.volume*, and thus the former inherits its constructors from the latter. This is similar to constructors of anonymous inner classes in Java. As a result, the subobject constructor call in *Radio* remains valid for *TeenagerRadio.volume*.

The second way to create the teenager radio is to change the declared superclass of the *volume* subobject, as shown in Fig. 30. Suppose that *MaxBoundedValue* is a subclass of *BoundedValue* in which *getValue* always returns the upper bound. Because the *volume* subobject now has a different declared superclass, the subobject constructor call for *volume* in *Radio* is no longer valid. Therefore, the constructor of *TeenagerRadio* must perform the subobject constructor call itself.

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {
    super(0,freq);
    subobject.volume(0,11);
  }
  subobject volume MaxBoundedValue<Integer>;
}
```

Fig. 30: The declared superclass is changed.

The subobject constructor call for *volume* in *TeenagerRadio replaces* the subobject constructor call of *volume* in *Radio*. The latter is no longer executed when initializing a *TeenagerRadio*. Instead, the call in *TeenagerRadio* is executed *at the moment the subobject constructor call of* Radio.volume *would have been executed*. This ensures that the subobject is still initialized when the code following the subobject constructor call in *Radio* is executed.

Initialization of Nested Refined Subobjects So far, we implemented *TeenagerRadio* by refining the *volume* subobject, which is a direct subobject of *Radio*. But since *BoundedValue* itself uses a *Property* subobject for its value, we can also refine the nested *value* subobject of *volume*.

The third way to implement *TeenagerRadio* is to override *getValue* in *volume.value*, as shown in Fig. 31. Because the declared superclasses of the *volume* and *volume.value* subobjects have not changed, no explicit subobject constructor calls are needed.

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {super(0,freq);}
  subobject volume {
    subobject value {
      Float getValue() = 11
    }
  }
}
```

Fig. 31: Nested refinement without changing the declared superclass.

The fourth and final way to implement *TeenagerRadio* is to change the declared superclass of *volume.value*. In the code in Fig. 32, class *Eleven* is a subclass of *Property<Integer>* that always returns 11 as its value. The superclass of the *value* subobject of the *volume* subobject is then redefined to *Eleven*. Therefore, a new subobject constructor call is required to initialize the *volume.value* subobject. In this specific case the subobject constructor call is optional because *Eleven* has a default constructor.

```
class TeenagerRadio extends Radio {
  TeenagerRadio(Float freq) {
    super(0,freq);
    // Optional: Eleven has a default constructor.
    subobject.volume.value();
  }
  subobject volume {subobject value Eleven;}
}
```

Fig. 32: Nested refinement changes the declared superclass.

The subobject constructor call for a nested subobject must be written in a constructor of the *outermost* class to avoid ambiguities. Otherwise, a constructor definition would have to be written inside the enclosing subobject body. But this could lead to the typical problems with initialization in a multiple inheritance hierarchy when that subobject is refined. Therefore, subobjects cannot contain constructors. If the host lan-

guage already supports multiple inheritance, this could be allowed, but we do not want to force this problem onto a host language with single inheritance.

If a subobject redefines multiple subobjects, an explicit subobject constructor call is required. That call is executed *the first time* one of the redefined subobjects would been initialized. Since there is always one most specific version of a subobject, there is always one subobject constructor call used to initialize the subobject, namely, the subobject constructor call that corresponds to the most specific version.

A remaining issue with subobject initialization is that the superclass constructor can rely on properties of the subobjects after they have been initialized. Therefore, there is a need to be able to specify these properties. If a subclass explicitly initializes a subobject, it must then ensure that these properties hold after the initialization of that subobject. A mechanism to define such contracts is a topic for future work.

4 Illustrations of Subobject-Oriented Programming

The radio example suffices to illustrate how subobject-oriented programming works, but it is deceptively simple. In this section, we illustrate the possibilities of subobject-oriented programming using classes in the JLo library. Section 4.1 presents the classes for defining associations. Section 4.2 shows how to reuse advanced graph algorithms by building graphs on top of associations.

4.1 Subobjects for Associations

The library of JLo, the subobject-oriented extension of Java, contains classes for uni- and bi-directional associations. A subobject is used for each navigable end of an association. The code in Fig. 33 shows a class diagram of the association classes and their most important methods. The corresponding code is shown in Fig 34. Most definitions are omitted, and some names have been abbreviated to save space. The association classes in the JLo library also use wildcards in the type arguments to increase the flexibility. Wildcards are omitted as they are not needed to illustrate the use of association subobjects. The top interface provides only the functionality to query an association. Class *Property* represents an encapsulated field, which is a unidirectional association. Similar classes are defined for sets and lists.

A bidirectional association end is connected to the object on its side of the association, and offers methods for registering and unregistering other bidirectional association ends. The *reg* and *unreg* methods keep the association in a consistent state, and make it possible to mix unary and n-ary association ends. The *reg* method of a unary end disconnects from its current connection (if any) using *unreg* and connects to the given association end. The *reg* method of an n-ary association end simply adds the given association end.

Abstract class *SingleBidi* represents unary bidirectional associations. The *Property* subobjects to store the object at its own end and the connected association end are private to hide their setters. The exported methods are still public. The *connect* and *disconnect* methods uses the *reg* and *unreg* methods to keep the association consistent. The

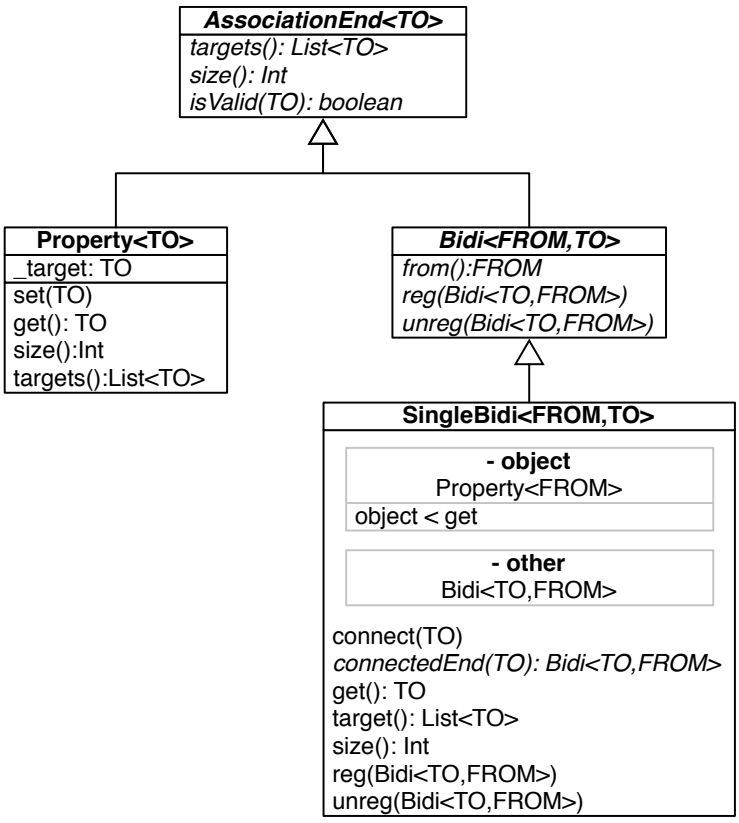


Fig. 33: A partial class diagram for the association classes.

abstract method *connectedEnd* determines the subobject at the other end of the association. It is implemented in the actual subobjects in the application. Similar classes are defined for n-ary bidirectional associations with set and list semantics. The library also provide classes for passive bidirectional associations, which do not provide a *connect* method because they are connected to different subobjects of different classes.

The class diagram in Fig. 35 and the code in Fig. 36 show how we can connect a microphone to a radio. The bidirectional association is implemented by simply specifying the association ends that must be connected. This is much simpler than writing the logic for keeping the association in a consistent state. Many programmers forget to clean up the back-pointers on at least one side of the association.

```
interface AssociationEnd<TO> {
 List<TO> targets();
 int size();
 boolean isValid(TO t);
}

class Property<TO> implements AssociationEnd<TO> {
 TO _target
 int size() = 1;
 void set(TO t) {if(isValid(t) {_target = t}}
 TO get() = _target
 List<TO> targets() = List(get())
}

interface Bidi<FROM,TO> extends AssociationEnd<TO> {
 FROM object();
 // internal bookkeeping methods
 void reg(Bidi<TO,FROM> b);
 void unreg(Bidi<TO,FROM> b);
}

abstract class SingleBidi<FROM,TO> implements Bidi<FROM,TO> {

 private subobject object Property<FROM> {
  export get as object;
 }
 private subobject other
         Property<Bidi<TO,FROM>>;

 TO get() = other.get().object()
 List<TO> targets() = List(get())
 int size() = 1;
 void connect(TO t) = {
  Bidi<TO,FROM> b = connectedEnd(t);
  reg(b);
  if (b != null) {b.reg(this.other);}
 }
 void reg(Bidi<TO,FROM> o) = {... other.set(o) ... }
 void unreg(Bidi<TO,FROM> o) = {... other.set(null) ... }
 abstract Bidi<TO,FROM> connectedEnd(TO t);
}
```

Fig. 34: Classes for association ends.

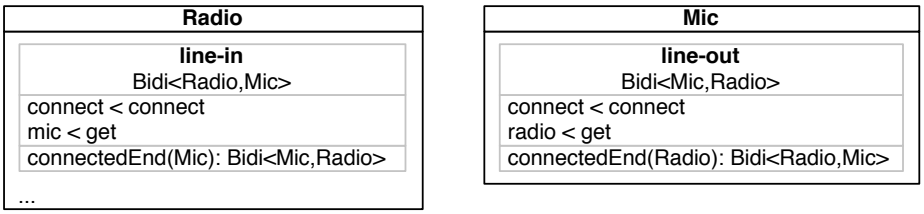


Fig. 35: Connecting a microphone to the radio.

```
class Radio {
  ...
  subobject line-in SingleBidi<Radio,Mic> {
    export connect, get as mic;
    Bidi<Mic,Radio> connectedEnd(Mic m) = m.line-out;
  }
}

class Mic {
  ...
  subobject line-out SingleBidi<Mic,Radio> {
    export connect, get as radio;
    Bidi<Radio,Mic> connectedEnd(Radio r) = r.line-in;
  }
}

Radio r = new Radio();
Mic m1 = new Mic();
Mic m2 = new Mic();
r.connect(m1);
assert(r.mic() == m1 && m1.radio() == r);
r.connect(m2);
assert(r.mic() == m2 && m2.radio() == r);
assert(m1.radio() == null);
```

Fig. 36: Connecting a radio to a microphone.

4.2 Subobjects for Graphs

The JLo library also contains classes to build weighted and unweighted graphs on top of the associations. The classes in the library allow advanced graph layouts, but for reasons of space we only present simple classes for homogeneous graphs.

```
abstract class DigraphNode<V> {
  abstract List<V> successors();
  abstract DigraphNode<V> node(V v);
  boolean isPredecessorOf(V v) = {...}
  List<V> allSuccessors() = {...}
  ...
}
```

Fig. 37: A class for graph nodes.

The code in Fig. 37 shows the top class of the graph library. The abstract *edges* method of *DigraphNode* must return the direct successor objects. The *node* method determines to which graph node of the direct successor this graph is connected. Based on this method, basic graph functionality can be implemented such as computing all successors of the current node.

The class diagram in Fig. 38 and the code in Fig. 39 show how graphs can be defined. A *Klass* has subobjects for its name, a list of superclasses, and a list of subobjects. A *Subobject* has subobjects for its name and its *Klass* (the declared superclass). A transitive association is used to define an association from a *Klass* to the declared superclasses of its subobjects. These associations are then used to build a type graph and an inheritance graph. The *isValid* methods of both *ListProperty* subobjects are overridden to forbid loops in the inheritance graph. Only simple configuration code was written, to obtain full graph functionality.

An alternative implementation could used graph nodes that support *AssociationEnd* subobjects as edges. In that case, *typeGraph* would be a graph node subobject that uses the *superKlasses* as its source of edges. For the inheritance graph, a *TransitiveAssociationEnd* subobject would be used and configured to use the *Klass.subobjects* and *Subobject.klass* subobjects. This transitive association subobject would then be used as a source of edges for the inheritance graph, together with *superKlasses*.

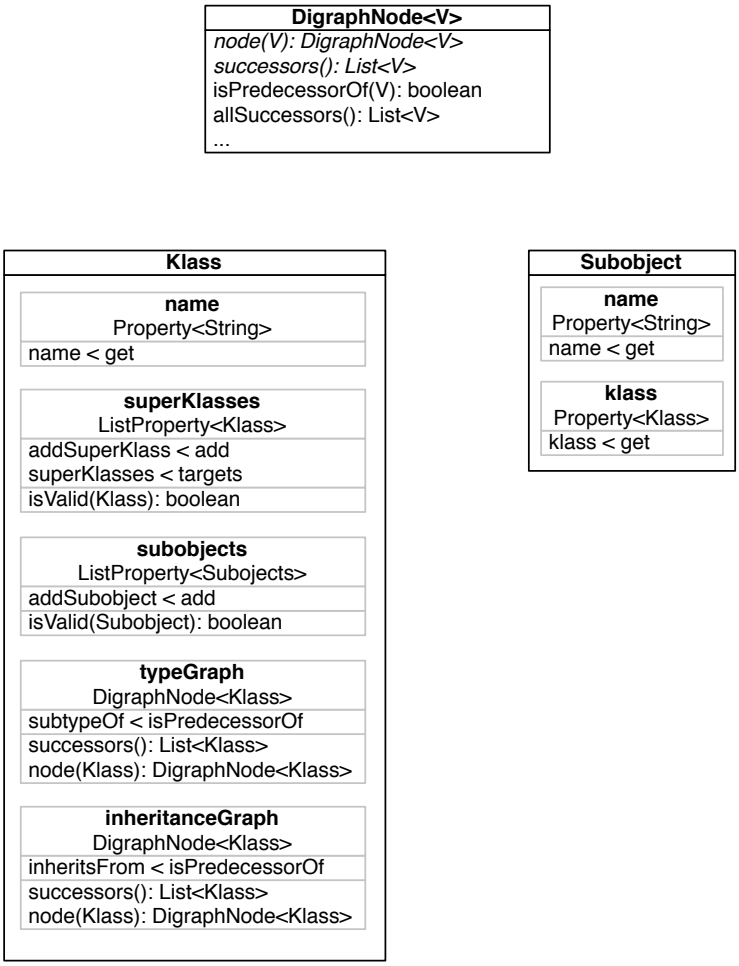


Fig. 38: A class diagram of class with graph subobjects.

```
class Klass {
  subobject name Property<String> {...}
  subobject superKlasses ListProperty<Klass> {
    export add as addSuperKlass,
         targets as superKlasses;
    boolean isValid(Klass klass) = !klass.inheritsFrom(outer);
  }
  subobject subobjects ListProperty<Subobject> {
    export add as addSubobject;
    boolean isValid(Subobject s) =
      !s.getKlass().inheritsFrom(outer);
  }

  // define graphs on top of the associations
  subobject typeGraph DigraphNode<Klass> {
    export isPredecessorOf as subtypeOf
    List<Klass> successors() = {
      // collect Klasses referenced by the subobjects
      ... subobject.getKlass() ...
    }
    DigraphNode<Klass> node(Klass klass) = klass.typeGraph;
  }

  subobject inheritanceGraph DigraphNode<Klass> {
    export predecessorOf as inheritsFrom
    List<Klass> successors() = {
      // collect Klasses referenced by the subobjects
      ... subobject.getKlass() ...
      // add the superklasses
      ... outer.superKlasses() ...
    }
    DigraphNode<Klass> node(Klass klass) = klass.inheritanceGraph;
  }
}
class Subobject {
  subobject name Property<String> {...}
  subobject klass Property<Klass> {...}
}
```

Fig. 39: Adding graph functionality with subobjects.

Weighted Graphs The JLo library also defines classes for weighted graphs, as shown in Fig. 40. Weighted graph node use explicit edges because each edge has its own weight. Class *WeightedNode* restricts the edges to weighted association ends such that it can offer additional functionality for weighted graphs. The *WeightedEnd* class represents a weighted association to objects of type *V* via intermediate objects of type *E*. The *first* and *second* methods return the *V* objects connected by the intermediate object such that *otherEnd* can compute the target objects of the association.

```
abstract class WeightedNode<V> implements DigraphNodeWithEdge<V> {
  abstract List<WeightedEnd<V,?>> edges();
  abstract WeightedNode<V> node(V v);
  Double shortestDistanceTo(V v) = {...}
  ...
}
abstract class WeightedEnd<V,I> implements AssociationEnd<V> {
  abstract List<I> intermediates();
  abstract Double weight(I i);
  abstract V first(I i);
  abstract V second(I i);
  V otherEnd(V v) =
     if(first(object()) == v) second(object())
     else first(object())
  ...
}
```

Fig. 40: Library classes for weighted graphs.

The use of the graph subobjects is illustrated in Fig. 41. A road has a length and is connected to two cities via bidirectional associations. The association ends in *Road* are implemented with subobjects *first* and *second*. Both are connected to *City.roads* by implementing *connectedEnd*.

Subobject *City.cityToCity* represents the weighted edge between two cities. Subobject *City.roadNetwork* implements the abstract methods of *WeightedNode* to select the edges and to select the graph node of the connected cities. It also exports the method to compute the shortest path to another city.

In a standard object-oriented style, where the graph structure is implemented with lots of low-level fields and methods, the graph algorithms would typically be reimplemented. Even if a graph library would be used, additional code would have to be written to make the graph structure visible for the graph library. In a subobject-oriented style, these structures are naturally available as defining them requires less effort than writing the corresponding low-level code.

```
class Road {
  Road(City first,City second,Double length) {
    subobject.first(this,first);
    subobject.second(this,second);
    subobject.length(length);
  }
  subobject length Property<Double> {
    export get as getLength;
  }
  subobject first SingleBidi<Road,City> {
    export connect as setFirst, target as getFirst;
    Bidi<City,Road> connectedEnd(City c) = c.roads
  }
  subobject second SingleBidi<Road,City> {
    export connect as setSecond, target as getSecond;
    Bidi<City,Road> connectedEnd(City c) = c.roads
  }
}

class City {
  City() {
    subobject.roads(this);
    subobject.roadNetwork(this);
  }
  subobject roads PassiveSetBidi<City,Road> {
    export targets as getRoads;
  }
  subobject cityToCity WeightedEnd<City,Road> {
    List<Road> intermediates() = outer.getRoads()
    Double weight(Road road) = road.getLength()
    City first(Road road) = road.getSecond()
    City second(Road road) = road.getFirst()
  }
  subobject roadNetwork SimpleWeightedNode<City> {
    export shortestDistanceTo as distanceTo;
    List<WeightedEnd<City,?>> edges() = List(outer.cityToCity)
    WeightedNode<City> node(City city) = city.roadNetwork
  }
}
```

Fig. 41: A subobject-oriented routing application.

5 Implementations

We have implemented subobject-oriented programming in two ways. The first implementation is a language extension of Java, called JLo, which is translated to Java code. The second implementation is a library for Python 3 that adds support for subobject-oriented programming by modifying objects and classes at run-time.

JLo [25] is a subobject-oriented extension of Java supported by an Eclipse plugin. The current implementation still uses Java syntax instead of the more concise Scala syntax used in the paper.The JLo compiler generates delegation code and wraps subobject constructor calls in Strategy objects. Special constructors are generated to allow subclasses to refine subobjects. The result is similar to the code in Fig. 5. To preserve multiple inheritance of subobjects, a JLo class is split into a Java interface and Java class. To enable super calls, methods are duplicated and given a unique name. Super calls are first resolved and then rewritten to regular invocations of the generated method that corresponds to the super method.

We also implemented a library [25] for subobject-oriented programming in Python 3. Support for redefining members in an enclosing scope is ongoing work. The library is implemented by two functions: **with_subobjects** and **subobject**, which are used to decorate classes. In Python, the decorated class definition *@expr **class** C: body* expands to *f=expr; **class** C: body ; C=f(C)*.

The @**subobject**($args$) decorator replaces the nested class with an instance of internal class *Subobject* that records $args$ and the class body. The @**with_subobjects** decorator collects the *Subobject* class members, and generates the delegation code. In addition, it adds a mk_s method for initializing the subobject. Finally, it initializes the ***self**.outer* field of the subobject to point to the outer object.

Fig. 42 shows how a subobject-oriented radio is implemented in Python. To define a subobject s in a class C, a nested class s is defined inside C. Class s is then decorated with **subobject**, and C is decorated with **with_subobjects**. To export members, a name mapping is passed as a set-valued argument with name *exports*. Finally, an object initializes its subobjects by calling the corresponding *mk_X* methods.

```
@with_subobjects
class Radio:
  @subobject(exports={'getValue':'getVolume',
                      'setValue':'setVolume'})
   class volume(BoundedValue): pass

  @subobject(exports={'getValue':'getFrequency',
                      'setValue':'setFrequency'})
   class frequency(BoundedValue): pass

  def __init__(self,vol,freq):
    self.mk_volume(0,vol,11)
    self.mk_frequency(87.5,freq,108)
```

Fig. 42: A subobject-oriented radio in Python.

6 Semantics of Subobjects

This section presents the core semantics of subobject-oriented programming by describing how dispatch works for a class-and-subobjects conglomerate in the presence of method and subobject aliasing, method overriding and further binding of subobjects. The semantics addresses two core issues: which method body a path resolves to in a given context; and which context the method body runs in. Given this information, a complete formal semantics for a language based on subobject-oriented programming can be designed in a now-standard fashion.

This semantics is useful, for instance, to help compiler writers correctly implement subobject-oriented programming dispatch mechanism. The semantics also forms the basis of the notion of ambiguity, which any implementation would need to check. If a path resolves to two different method bodies, then a class is ambiguous and an explicit declaration is required to resolve the conflict. Finally, the semantics could form the basis of a type-theoretic foundation of subobject-oriented programming, but this is left for future work.

The following conventions will be used in the semantics: m denotes a method name; t is a class/subobject name. Paths, P, are defined by the following grammar:

$$\begin{aligned} P &::= t \mid \mathsf{this} \mid \mathsf{super} \mid \mathsf{outer} \mid P.P \\ M &::= P.m \\ A &::= \epsilon \mid A.t \end{aligned}$$

P is a path ending in a class or subobject name. M is a path ending in a method name. Paths may also include this, super and outer, where this refers to the current dynamic class/subobject, super refers to the superclass, and outer refers to the surrounding class/subobject. A path is *pure* if it contains no occurrence of super, this, or outer. A, B, C denote *absolute paths*, consisting only of class/subobject names. Absolute paths refer to locations in code and are therefore used to uniquely identify classes and subobjects.

The semantics is based on several judgements (Fig. 43) that capture the essence of method, subobject and aliasing declarations. These play the role of axioms in the formal system and specific instances can easily be derived from the code. Relations of the form $\Im \in_d A$ capture that some declaration $\Im$ is made in class A. In contrast, relations $\Im \in^* A$ will be introduced later to capture all the declared and inherited (but not overridden) facts about class/subobject A. Aliasing clauses encode the implicit relationship introduced via an **export** clause or through named parameters. For example, a clause **export** a **as** b appearing in subobject P within the context of class/subobject A is modelled by axiom b aliases $P.a \in_d A$.

Both m aliases $M \in_d A$ and t aliases $P \in_d A$ have restrictions. Paths M can be of the form $t.m$, for exporting a method of a subobject into the current interface. This will ensure that the method referred to is one in a direct subobject. The path P is of the form $t'.t''$ to ensure that the aliased subobject is a directly nested subobject of the class/subobject where the declaration occurs. Cases $t \in_d \epsilon$ and t subclasses $t' \in_d \epsilon$ indicate that the declaration is made at the top level, thus, in this case, t and t' are classes and t subclasses t'. For an overriding clause, m overrides $M \in_d A$, M can only be a pure path,

$m \mapsto b \in_d A$	method m with body b
m aliases $M \in_d A$	aliasing of m and M
$t \in_d A$	class/subobject t
t aliases $P \in_d A$	aliasing of t and P
t subclasses $t' \in_d A$	subclassing or subobject typing
m overrides $M \in_d A$	overriding of method M

Fig. 43: Judgements: Axiom schemes encoding explicit declarations ($- \in_d A$) in class/-subobject A.

and m must be a declared or inherited method or an alias to a method, otherwise it is an error.

The following predicates, which can be trivially computed based on the axioms above, will be useful.

$$
\begin{aligned}
m \text{ not declared in } A &\mathrel{\widehat{=}} \neg(\exists b \cdot m \mapsto b \in_d A) \\
t \text{ not declared in } A &\mathrel{\widehat{=}} \neg(t \in_d A) \\
m \text{ not aliased in } A &\mathrel{\widehat{=}} \neg(\exists M \cdot m \text{ aliases } M \in_d A) \\
t \text{ not aliased in } A &\mathrel{\widehat{=}} \neg(\exists P \cdot t \text{ aliases } P \in_d A) \\
m \text{ no new binding } A &\mathrel{\widehat{=}} m \text{ not declared in } A \ \wedge \ m \text{ not aliased in } A
\end{aligned}
$$

The remainder of the semantics will be presented in three interdependent fragments, expressing the inheritance relationships between classes and subobjects (Section 6.1), expressing what is inherited (Section 6.2), and expressing dispatch by resolving path expressions (Section 6.3).

6.1 Inheritance

Next we define a set of rules capturing the inheritance relationship between various classes and subobjects. There are two paths to inheritance: directly via subclassing and indirectly when (potentially) further binding an inherited subobject. The subclassing and inheritance relationships is kept separate, as subclassing is needed for resolving super. The inheritance relation is intransitive and is defined by the two judgements:

A subclasses B	A subclasses B
A inherits B	A inherits from B

These judgements are defined globally using absolute paths, rather than being defined within the context of a specific class (Fig. 44). The first rule for inherits converts subclassing to inheritance. The second rule captures inheritance of subobjects.

6.2 Class/Subobject Contents

The judgements in Fig. 45 describe the contents of a class or subobject, including what is inherited and derived. Method bodies and subobjects also record the location of

$$\frac{t \text{ subclasses } t' \in_d A}{A.t \text{ subclasses } t'} \qquad \frac{P \text{ subclasses } P'}{P \text{ inherits } P'} \qquad \frac{P \text{ inherits } P' \quad (t, \_) \in^* P'}{P.t \text{ inherits } P'.t}$$

Fig. 44: Rules: Subclassing and inheritance.

the corresponding declaration as an absolute path. The judgements for $\downarrow$overrides and overrides$\downarrow$ are used for resolving the left-hand side and the right-hand side of an overriding clause. The first is used to determine the location of the overriding method and the second is used to find the end of the alias chain that will dispatch to it. The judgement for dispatches to gives the methods actually available after overriding and inheritance, plus an adjustment to move the 'this' pointer to the correct location within class-and-subobjects conglomerate. FIXEXPLAIN—REVIEWER COMMENT. WHAT IS ROLE of P in dispatcehs to.

$(t, B) \in^* A$	subobject t from source B
$m \mapsto (b, B) \in^* A$	method m with body b from B
m aliases $M \in^* A$	aliasing of m and M
t aliases $P \in^* A$	aliasing of t and P
$M \downarrow$overrides $M' \in^* A$	resolution of M in overriding
(M, b, B) overrides$\downarrow M' \in^* A$	relocating method to M'
m dispatches to $(b, B, P) \in^* A$	dispatch candidate for m. P is used to adjust the dynamic pointer

Fig. 45: Judgements: Declared and inherited class/subobject contents for class A. FIX: These descriptions are terrible.

The judgements in Fig. 46 describe the rules for declared and inherited classes/subobjects and subobject aliasing. In the last rule, t not declared in B prevents subobject t being declared in B. Permitting it would allow non-local further binding, resulting in ambiguity as subobjects could be further bound in more than one code location.

$$\frac{t \in_d A}{(t, A.t) \in^* A} \qquad \frac{(t, P) \in^* A \quad B \text{ inherits } A \quad t \text{ not declared in } B}{(t, P) \in^* B}$$

$$\frac{t \text{ aliases } P \in_d A}{t \text{ aliases } P \in^* A} \qquad \frac{t \text{ aliases } P \in^* A \quad B \text{ inherits } A \quad t \text{ not declared in } B}{t \text{ aliases } P \in^* B}$$

Fig. 46: Rules: Classes/subobjects and subobject aliasing

The rules in Fig. 47 describe method aliasing. An aliasing declaration is inherited even when a new method is declared, in which case the new method also overrides

the aliased method. Recall that when two method paths are aliased, they can never be broken apart.

$$\frac{m \text{ aliases } M \in_d A}{m \text{ aliases } M \in^* A} \qquad \frac{m \text{ aliases } M \in^* A \quad B \text{ inherits } A}{m \text{ aliases } M \in^* B}$$

Fig. 47: Rules: Method aliasing

Based on the previous rules, define the following:

$$\text{not aliased } t \in^* A \mathrel{\widehat{=}} \neg(\exists P \cdot t \text{ aliases } P \in^* A)$$
$$\text{not aliased } m \in^* A \mathrel{\widehat{=}} \neg(\exists M \cdot m \text{ aliases } M \in^* A)$$

The most complicated set of rules deal with the interaction between overriding declarations and aliasing. The first collection of rules (Fig. 48) initiates the resolution process based on the declared and inherited overriding declarations. The second collection of rules (Fig. 49) deal with finding an appropriate method body by resolving the LHS of an overrides clause. The third collection of rules (Fig. 50) 'move' this method to the place being overridden, resolving any aliasing along the way. This sets things up so that when performing path resolution to dispatch a method call, one simply follows an alias chain until the end.

$$\frac{m \text{ overrides } M \in_d A}{m{\downarrow}\text{overrides } M \in^* A} \qquad \frac{\begin{array}{c} m \text{ overrides } M \in^* A \quad B \text{ inherits } A \\ m \text{ no new binding } B \end{array}}{m{\downarrow}\text{overrides } M \in^* B}$$

Fig. 48: Rules: Initiate overriding resolution

The first rule in Fig. 49 resolves aliasing of m on the left hand side, if no method is found at M. The second rule deals with a method path starting with a subobject name that is not aliased: the search moves into the subobject. The third rule deals with the case that the subobject name is aliased. The fourth and fifth rules switch to resolving the right-hand side when a candidate method body is found.

The first rule in Fig. 50 removes any `outer` added in the previous phase and the second rule removes any t from the right-hand side, both adjusting the P component; the adjustments will be used to move the dynamic pointer from the end of the alias chain back to where the method is declared. The third rule expands a subobject alias. The fourth rule expands a method alias.

The rules in Fig. 51 deal with the ultimate dispatch candidates for simple paths consisting of a single name m in the context of some class/subobject. The three cases are when a method is declared in the class/subobject, when a method is inherited but not overridden in any way, and when some external (to the class/subobject) overriding declaration is present.

$$\frac{m{\downarrow}\textsf{overrides}\ M \in^* A \quad m\ \textsf{aliases}\ M' \in^* A \quad m\ \textsf{not declared in}\ A}{M'{\downarrow}\textsf{overrides}\ M \in^* A} \qquad \frac{t.M'{\downarrow}\textsf{overrides}\ M \in^* A \quad \textsf{not aliased}\ t \in^* A}{M'{\downarrow}\textsf{overrides outer}.M \in^* A.t}$$

$$\frac{t.M'{\downarrow}\textsf{overrides}\ M \in^* A \quad t\ \textsf{aliases}\ P \in^* A}{P.M'{\downarrow}\textsf{overrides}\ M \in^* A} \qquad \frac{m{\downarrow}\textsf{overrides}\ M \in^* A \quad m \mapsto (b, B) \in^* A}{(b, B, \epsilon)\ \textsf{overrides}{\downarrow} M \in^* A}$$

$$\frac{m\ \textsf{aliases}\ M \in_d A \quad m \mapsto (b, B) \in^* A}{(b, B, \epsilon)\ \textsf{overrides}{\downarrow} M \in^* A}$$

Fig. 49: Rules: Non-local overriding—finding method body

$$\frac{(b, B, P)\ \textsf{overrides}{\downarrow}\textsf{outer}.M' \in^* A.t}{(b, B, t.P)\ \textsf{overrides}{\downarrow} M' \in^* A}$$

$$\frac{(b, B, P)\ \textsf{overrides}{\downarrow} t.M \in^* A \quad \textsf{not aliased}\ t \in^* A}{(b, B, \textsf{outer}.P)\ \textsf{overrides}{\downarrow} M \in^* A.t}$$

$$\frac{(b, B, P)\ \textsf{overrides}{\downarrow} t.M \in^* A \quad t\ \textsf{aliases}\ P' \in^* A}{(b, B, P)\ \textsf{overrides}{\downarrow} P'.M \in^* A}$$

$$\frac{(b, B, P)\ \textsf{overrides}{\downarrow} m \in^* A \quad m\ \textsf{aliases}\ M \in^* A}{(b, B, P)\ \textsf{overrides}{\downarrow} M \in^* A}$$

Fig. 50: Rules: Non-local overriding—moving to end of alias chain

$$\frac{m \mapsto b \in_d A}{m\ \textsf{dispatches to}\ (b, A, \epsilon) \in^* A}$$

$$\frac{m \mapsto (b, A) \in^* A \quad B\ \textsf{inherits}\ A \quad m\ \textsf{no new binding}\ B}{m\ \textsf{dispatches to}\ (b, A, \epsilon) \in^* B}$$

$$\frac{(b, B, P)\ \textsf{overrides}{\downarrow} m \in^* A \quad \textsf{not aliased}\ m \in^* A}{m\ \textsf{dispatches to}\ (b, B, P) \in^* A}$$

Fig. 51: Rules: Dispatch candidates

The diagram in Fig. 52 illustrates the results of applying the rules for $\downarrow$overrides and overrides$\downarrow$. Assume that the facts derived from code are m aliases $t_1.t_2.t_3.n \in^* A$, and m overrides $s_1.s_2.s_3.p \in_d A$, where the first fact would be derived from m aliases $t_1.n \in_d A$, and n aliases $t_2.n \in_d A.t_1$, n aliases $t_3.n \in_d A.t_1.t_2$.

The first intermediate result is

$$n{\downarrow}\mathsf{overrides}\ \mathsf{outer.outer.outer}.s_1.s_2.s_3.p \in^* A.t_1.t_2.t_3,$$

giving the result after resolving the aliasing, thus n will be the name of some actual method with body b_n declared in some class/subobject B. From this it immediately follows that:

$$(b_n, B, \epsilon)\ \mathsf{overrides}{\downarrow}\mathsf{outer.outer.outer}.s_1.s_2.s_3.p \in A.t_1.t_2.t_3.$$

The second intermediate result is

$$(b_n, B, \mathsf{outer.outer.outer}.t_1.t_2.t_3)\ \mathsf{overrides}{\downarrow}p \in^* A.s_1.s_2.s_3$$

which gives relates the overriding method body (b_n, B) with the place where the overriding occurs, namely, $A.s_1.s_2.s_3.p$. The resulting dispatch candidate will be

$$p\ \mathsf{dispatches\ to}\ (b_n, B, \mathsf{outer.outer.outer}.t_1.t_2.t_3) \in^* A.s_1.s_2.s_3.$$

The additional component, $\mathsf{outer.outer.outer}.t_1.t_2.t_3$, is applied to a dynamic pointer during dispatch to move it to the correct location (following the dotted arrow in Fig. 52).

6.3 Path Resolution

The semantics of path resolution is based on three judgements (Fig. 53). The judgements describe how to lookup a method body in a given context, how to evaluate a method body in a given context, and how to resolve a path to a subobject.

Resolving a path involves navigating around a class and its respective subobjects and their superclasses. Doing so requires keeping track of two 'pointers', one for the dynamic class/subobject of 'this', the other for the static code location where the current class/subobject is found. This will be represented by $\langle D, S\rangle$, where D is the dynamic part and S is the static part of the location.

The rules in Fig. 54 deal with method dispatch. We assume that local method call paths begin with this, as it plays the role in the semantics to ensure that the most specific static class/subobject is considered. All other components of the dispatch is done based on the static pointer. The first rule finds the candidate method associated with the equivalence class of paths to the method. A path of the form $D.\mathsf{outer}^n.P'$, introduced when P is appended to D in the first rule, can be reduced to an absolute path by iterating the following equivalence $D.t.\mathsf{outer}.P = D.P$ until all occurrences of outer have been eliminated. The rules for evaluating a method body have been omitted, but can be added in a straightforward fashion; the key point of interest is that the context in which that body is run is, namely, some pair $\langle D, S\rangle$.

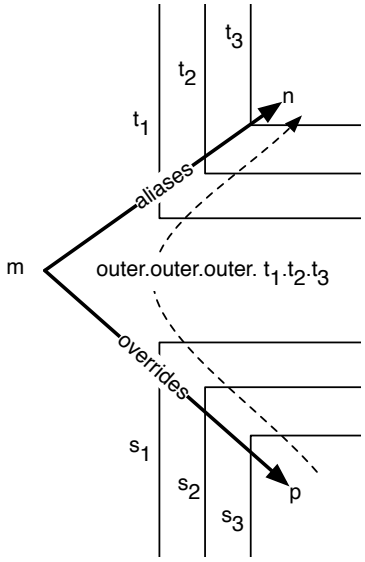


Fig. 52: Example: Overriding resolution, assuming m overrides $s_1.s_2.s_3.p$ and m overrides $s_1.s_2.s_3.p$ are declared in A.

$\langle D, S\rangle\ M \Longrightarrow v$	method M evaluates to v
$\langle D, S\rangle\ b \Longrightarrow v$	body b evaluates to v (omitted)
$\langle D, S\rangle \xrightarrow{P} \langle D', S'\rangle$	P moves from $\langle D, S\rangle$ to $\langle D', S'\rangle$

Fig. 53: Judgements: Resolution and evaluation — D, S are absolute paths

$$\frac{m \text{ dispatches to } (b, B, P) \in^* S \quad \langle D.P, B\rangle\ b \Longrightarrow v}{\langle D, S\rangle\ m \Longrightarrow v}$$

$$\frac{\langle D, S\rangle \xrightarrow{P} \langle D', S'\rangle \quad \langle D', S'\rangle\ m \Longrightarrow v}{\langle D, S\rangle\ P.m \Longrightarrow v}$$

Fig. 54: Rules: Method path resolution

$$\frac{D = A.t \qquad (t,P) \in^* A}{\langle D,S\rangle \xrightarrow{\mathsf{this}} \langle D,P\rangle} \qquad \frac{(t,P) \in^* S}{\langle D,S\rangle \xrightarrow{t} \langle D.t,P\rangle}$$

$$\frac{t \text{ aliases } P \in^* S \quad \langle D,S\rangle \xrightarrow{P} \langle D',S'\rangle}{\langle D,S\rangle \xrightarrow{t} \langle D',S'\rangle} \qquad \frac{S \text{ subclasses } S'}{\langle D,S\rangle \xrightarrow{\mathsf{super}} \langle D,S'\rangle}$$

$$\frac{}{\langle D.t,S.t'\rangle \xrightarrow{\mathsf{outer}} \langle D,S\rangle} \qquad \frac{\langle D,S\rangle \xrightarrow{P} \langle D'',S''\rangle \quad \langle D'',S''\rangle \xrightarrow{P'} \langle D',S'\rangle}{\langle D,S\rangle \xrightarrow{P.P'} \langle D',S'\rangle}$$

Fig. 55: Rules: Path Resolution.

The rules for path resolution are given in Fig. 55. Each part of a path results in an incremental change to both the dynamic and static parts of the context. The rules are ambiguous in the case that a subobject and an alias with the same name coexist together. The compiler must rule out such cases. The rule for this selects most specific textual class for a given dynamic class to start the search. The two rules for subobject name t look up the name in the given static path or replace it with the path it aliases, respectively. The rule for super finds the superclass of the current static code location, and the rule for outer finds the surrounding class/subobject of the current static code location. The final rule describes how to resolve longer paths.

These rules capture the essence of the composition mechanism of subobject-oriented programming. Providing a complete semantics for the full language including type safety theorems and their proof is a topic for future work.

7 Related Work

For a discussion on aspect-oriented programming, mixins, traits, and non-conformant inheritance, we refer to Section 2.

Subobject-oriented programming is based on the *component relation* that we previously introduced for composition of classes [26]. Subobjects refine the component relation in a number of ways. With the component relation, member redefinitions were written in the composed class, and then wired into the subobject with overrides clauses. This lead to scattering of subobject members throughout the composed class. With subobjects, such redefinitions can be written directly in the subobject. This eliminates the overrides clauses, and significantly improves readability. The dedicated parameter mechanism for connecting components was removed and replaced by using methods to connect subobjects to each other. Switching to Scala syntax for method definitions removed most of the overhead. Most importantly, our previous work lacked support for object initialization and super calls, both of which are essential in real programming languages. In addition, our previous work was not implemented and demonstrated only very basic subobjects.

Reppy and Turon present trait-based metaprogramming [19], which is very similar to non-conformant inheritance in Eiffel. Traits are checked at compile-time and then inlined. The name of each method is a parameter that is used to rename it in the reusing

class. Because the renaming is deep, this allows proper resolution of name conflicts, but it also requires a lot of work. If two traits of the same kind are used, all of their methods must either be renamed or excluded. Contrary to Eiffel, sharing is not the default policy. Instead, a type error is reported when multiple methods with the same signature are inlined. A trait can contain fields, which must be initialized by the constructor of a class that uses the trait. The special type *ThisType* is used to impose constraints on the class that reuses a trait. This is similar to requirements in regular traits, and to abstract methods in subobjects. A method of a trait can override a method of the outer class. The original method is available as *outer.m(...)*. Once a trait method overrides a method of the outer class, it is considered to be locally defined. As a result, it can again be overridden by another trait method. The resulting concatenation of traits is similar to mixin-based inheritance. Contrary to subobjects, traits cannot be used as separate objects, prevent certain kinds of reuse.

Object layout in C++ [22] is often described in terms of subobjects, where each inherited class forms a subobject. The key difference with our subobjects is that in our approach subobjects are placed in separate namespaces, which avoids many of the problems of C++. More concisely, C++ implements subobject-based inheritance, whereas subobject-oriented programming is about composition of subobjects. Refinement of subobjects is not possible in C++. Our approach uses the rule-of-dominance of C++ to resolve conflicts if a single best candidate is available.

Languages that implement further binding include Beta [17], gbeta [10], and Tribe [5]. In these languages, further binding applies to nested classes, which can be used to create objects of the same family. In our approach, further-binding applies to nested subobjects, which define a static part of the composed class. Virtual classes [9, 11] do resemble subobjects, but their purpose is completely different. Virtual classes support family polymorphism, whereas subobjects support composition of classes. As such, neither technique can be used as a substitute for the other. Any number of objects can be constructed from virtual classes and path-dependent types are required to ensure that certain object belong to the same family. With subobjects on the other hand, there is only one "instance" of each subobject per outer object. Subobject names serve only to avoid conflicts and access the parts. Therefore, path-dependent types and the associated complexities are not needed.

Subject-oriented programming [13] differs from subobject-oriented programming in the purpose of the composition. Composition in subject-oriented programming is about the separation of concerns, and thus more related to aspects-oriented programming and family polymorphism, whereas subobject-oriented programming is about composing classes, and is thus more of a refinement of classical object-oriented programming. Both approaches are complementary, since the different view-points could implement parts of their customization with subobjects.

Madsen and Møller-Pedersen [16] introduced part objects in the context of Beta for better structuring code. A part object is a locally defined object. Part objects know their location, which in our setting is given by the **`outer`** reference. Their motivation is similar to ours, but their language lacks the constructs for composing and refining the part objects (subobjects) as ours does—more precisely, these need to be coded up

in regular Beta code. Beta does however also offer refinement/further binding of nested classes, which is something few other languages support.

A split object [2] consists of a collection of *pieces* which represent particular viewpoints or roles of the split object, have no identity, and are organized in a delegation hierarchy. Invoking methods is done by selecting a viewpoint to send the message to. The main difference with subobjects is that subobjects are used to build classes, whereas pieces are used to model different viewpoints on a class. The substructures in both approaches have an opposite order with respect to overriding. A piece overrides methods from its enclosing pieces and class, whereas enclosing subobjects and the composed class override methods of more deeply nested subobjects. In addition, members in pieces cannot be merged, whereas members from different subobjects can be merged. Finally, pieces are added dynamically, whereas subobjects are declared statically.

Blake and Cook [3] propose to add part hierarchies to object-oriented languages. These resemble nesting of subobjects, but the proposed implementation does not include the advanced features for refining subobjects.

8 Conclusion

Existing object-oriented and aspect-oriented techniques do not offer the features to build a class using other classes as building blocks. Instead of being encapsulated in a class and reused, cross-cutting structural code for general purpose concepts such as associations and graphs must be implemented over and over again.

Subobject-oriented programming improves on object-oriented programming by allowing programmers to easily build a class from other classes. This work improved on our previous work in a number of ways. We defined subobject initialization and super calls. We improved the adaptability of subobjects by using refinement instead of overriding. We improved the readability of subobjects using a more object-oriented syntax and removed the functional style parameter mechanism. In addition, we have implemented subobject-oriented programming as a language extension to Java [12], and as a library in Python 3 [20]. Finally, we have also developed a library of classes that demonstrates the advanced possibilities of subobject-oriented programming.

References

1. I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of caesarj. In A. Rashid and M. Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 135–173. 2006.
2. D. Bardou and C. Dony. Split objects: a disciplined use of delegation within objects. In *Proceedings of OOPSLA '96*, pages 122–137. ACM Press, 1996.
3. E. H. Blake and S. Cook. On including part hierarchies in object-oriented languages with an implementation in smalltalk. In *Proceedings of ECOOP '87*, pages 41–50, 1987.
4. G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOOP '90*, pages 303–311, 1990.
5. D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: a simple virtual class calculus. In *Proceedings of AOSD '07*, pages 121–134, 2007.

6. D. Colnet, G. Marpons, and F. Merizen. Reconciling subtyping and code reuse in object-oriented languages: Using inherit and insert in SmartEiffel, the GNU Eiffel compiler. In *ICSR*, 2006.
7. T. V. Cutsem, A. Bergel, S. Ducasse, and W. De Meuter. Adding state and visibility control to traits using lexical nesting. In *ECOOP*, pages 220–243, 2009.
8. S. Ducasse, R. Wuyts, A. Bergel, and O. Nierstrasz. User-changeable visibility: resolving unanticipated name clashes in traits. In *OOPSLA*, pages 171–190, 2007.
9. E. Ernst. Family polymorphism. In *ECOOP*, pages 303–326, 2001.
10. E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings of ECOOP '03*, LNCS 2743, pages 303–329. Springer-Verlag, July 2003.
11. E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL*, pages 270–282, 2006.
12. J. Gosling et al. *The Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 2000.
13. W. H. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA '93*, pages 411–428, 1993.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *ECOOP '01*, pages 327–354.
15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. *ECOOP*, pages 220–242, 1997.
16. O. L. Madsen and B. Møller-Pedersen. Part objects and their location. In *Proceedings of TOOLS '92*, pages 283–297, 1992.
17. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.
18. M. Odersky and M. Zenger. Scalable component abstractions. In *Proceedings of OOPSLA '05*, pages 41–57, 2005.
19. J. H. Reppy and A. Turon. Metaprogramming with traits. In *Proceedings of ECOOP '07*, pages 373–398, 2007.
20. G. V. Rossum and F. Drake. *Python 3 Reference Manual*. CreateSpace, 2009.
21. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP*, pages 248–274, 2003.
22. B. Stroustrup. *The C++ programming language (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.
23. Technical Group 4 of Technical Committee 39. *ECMA-367 Standard: Eiffel Analysis, Design and Programming Language*. ECMA International, 2005.
24. M. van Dooren and D. Clarke. Subobject transactional memory. In *COORDINATION*, page to appear, 2012.
25. M. van Dooren and B. Jacobs. Implementations of subobject-oriented programming, 2011. `http://people.cs.kuleuven.be/marko.vandooren/subobjects.html`.
26. M. van Dooren and E. Steegmans. A higher abstraction level using first-class inheritance relations. In *ECOOP*, pages 425–449, 2007.